ASSIGNMENT 6

1. BFS

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    result = []

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            result.append(node)
            queue.extend(graph[node])

    return result


# Example usage:
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}
start_node = 'A'
print("BFS:", bfs(graph, start_node))
```

2.DFS

```
def dfs(graph , node, visited = None):
    if visited is None:
        visited = set()
    visited.add(node)
    print(node , end = ' ')

    for neighbour in graph[node]:
        if neighbour not in visited:
            dfs(graph, neighbour , visited)
```

```
graph = {
    'A' : ['B' , 'C'],
    'B' : ['A' , 'D' , 'E'],
    'C' : ['A' , 'F'],
    'D' : ['B'] ,
    'E' : ['B' , 'F'],
    'F' : ['E' , 'C']
}

dfs(graph , 'A')
```

3.Tic-tac-toe

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
    print()


def check_winner(board):
    for row in board:  # Check rows
        if row[0] == row[1] == row[2] != ' ':
            return row[0]
    for col in zip(*board):  # Check columns
        if col[0] == col[1] == col[2] != ' ':
            return col[0]
    if board[0][0] == board[1][1] == board[2][2] != ' ':  # Check
diagonals
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return board[0][2]
    return None


def tic_tac_toe():
    board = [[' ']*3 for _ in range(3)]
    players = ['X', 'O']
    for turn in range(9):
        print_board(board)
        player = players[turn % 2]
        print(f"Player {player}'s turn")
```

```
        row, col = map(int, input("Enter row and column (0-2): 
").split())
        if board[row][col] == ' ':
            board[row][col] = player
            winner = check_winner(board)
            if winner:
                print_board(board)
                print(f"Player {winner} wins!")
                return
        else:
            print("Cell already occupied, try again!")
            turn -= 1
    print_board(board)
    print("It's a draw!")


tic_tac_toe()
```

ASSIGNMENT 7

1. N Queen

```
def solve_n_queens(n):
    def solve(row):
        if row == n:
            solutions.append(["." * c + "Q" + "." * (n - c - 1) for c 
in board])
        for col in range(n):
            if all(board[i] != col and abs(board[i] - col) != row - i 
for i in range(row)):
                board[row] = col
                solve(row + 1)

    solutions, board = [], [-1] * n
    solve(0)
    return solutions

n = 4  # Change to 8 for 8-Queen problem
for solution in solve_n_queens(n):
    print("\n".join(solution) + "\n")
```

2. 8 Puzzle problem

```
from heapq import heappush , heappop

def a_star(start,goal):
    def h(state):
        return sum(abs(state.index(i) % 3 - goal.index(i) % 3) + abs(state.index(i) // 3 - goal.index(i) // 3) for i in range(1,9))

    def get_neighbours(state):
        idx = state.index(0)
        neighbours = []
        moves = [(-1,0) , (1,0) , (0,1) , (0,-1)]
        for dx,dy in moves:
            x , y = idx % 3 , idx // 3
            nx , ny = x + dx , y + dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                nidx = ny * 3 + nx
                neighbour = state[:]
                neighbour[idx] , neighbour[nidx] = neighbour[nidx] , neighbour[idx]
                neighbours.append(neighbour)
        return neighbours

    frontier = []
    heappush(frontier , (h(start) , start, 0 , []))
    visited = set()

    while frontier:
        _,current,cost,path = heappop(frontier)
        if tuple(current) in visited:
            continue
        path = path + [current]
        if current == goal:
            return path
        visited.add(tuple(current))
        for neighbour in get_neighbours(current):
            heappush(frontier , (h(neighbour) + cost , neighbour, cost +1, path))
    return None
```

```
start = [1,2,3,4,0,5,6,7,8]
goal  = [1,2,3,4,5,6,7,8,0]

solution = a_star(start,goal)
if solution:
    for step in solution:
        for i in range(0,9,3):
            print(step[i : i+3])
        print()
else:
    print("No solution")
```

ASSIGNEMNT 8

1.A * Algorithm

```
from heapq import heappush , heappop
def a_star_search(graph , start , goal , heuristic):
    pq , visited = [(0 + heuristic[start] , 0 , start , [])] , set()
    while pq:
        _,cost,node,path = heappop(pq)
        if node in visited:
            continue
        path = path + [node]
        if node == goal:
            return path
        visited.add(node)

        for neighbour , weight in graph[node]:
            if neighbour not in visited:
                heappush(pq, (weight + cost + heuristic[neighbour] ,
weight + cost , neighbour , path))

    return None

graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 1), ('E', 4)],
    'C': [('F', 2)],
    'D': [], 'E': [], 'F': []
}
heuristic = {'A': 6, 'B': 4, 'C': 4, 'D': 0, 'E': 2, 'F': 0}
```

```
print(a_star_search(graph , 'A' ,'D' , heuristic))
```

2. AO*

```
class Node:
    def __init__(self, name, heuristic=0):
        self.name = name
        self.heuristic = heuristic
        self.successors = []

    def add_successor(self, cost, *nodes):
        self.successors.append((cost, nodes))

def ao_star(node):
    if not node.successors:
        return node.heuristic

    min_cost = float('inf')
    for cost, successors in node.successors:
        total_cost = cost + sum(ao_star(s) for s in successors)
        if total_cost < min_cost:
            min_cost = total_cost

    node.heuristic = min_cost
    return min_cost

# Example graph
A = Node('A', 999)
B = Node('B', 4)
C = Node('C', 3)
D = Node('D', 2)
E = Node('E', 0)
F = Node('F', 0)

A.add_successor(5, B, C)
B.add_successor(2, D)
C.add_successor(3, E, F)

print("Optimal cost:", ao_star(A))
```

3.Hill Climbing

```
def hill_climbing(start, goal, h, graph):
    current = start
    while current != goal:
        neighbors = graph[current]
        next_node = min(neighbors, key=lambda x: h[x], default=None)
        if next_node is None or h[next_node] >= h[current]:
            break
        print(f"Move from {current} to {next_node}")
        current = next_node
    return current == goal


# Example graph
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': [],
    'F': []
}
h = {'A': 6, 'B': 4, 'C': 5, 'D': 1, 'E': 2, 'F': 0}  # Heuristic
values
start = 'A'
goal = 'F'


if hill_climbing(start, goal, h, graph):
    print("Reached the goal!")
else:
    print("Stuck in a local maximum!")
```

ASSIGNMENT 9

1.Tower of Hanoi

```
def tower_of_hanoi(n , source , target , auxiliary):
    if n == 1:
```

```
        print(f"Move disk 1 from {source} to {target}")
        return
    tower_of_hanoi(n-1 , source , auxiliary , target)
    print(f"Move disk {n} from {source} to {target}")
    tower_of_hanoi(n-1 , auxiliary , target , source)
n = 3
tower_of_hanoi(n, 'A' , 'C' , 'B')
```

2.Min max

```
def minimax(position, depth, is_maximizing, alpha=None, beta=None):
    if depth == 0 or is_terminal(position):
        return evaluate(position)

    if is_maximizing:
        max_eval = float('-inf')
        for child in get_children(position):
            eval = minimax(child, depth - 1, False, alpha, beta)
            max_eval = max(max_eval, eval)
            if alpha is not None:
                alpha = max(alpha, eval)
                if beta is not None and beta <= alpha:
                    break  # Alpha-Beta pruning
        return max_eval
    else:
        min_eval = float('inf')
        for child in get_children(position):
            eval = minimax(child, depth - 1, True, alpha, beta)
            min_eval = min(min_eval, eval)
            if beta is not None:
                beta = min(beta, eval)
                if alpha is not None and beta <= alpha:
                    break  # Alpha-Beta pruning
        return min_eval


# Tic-Tac-Toe specific helper functions
def is_terminal(board):
    return evaluate(board) != 0 or all(cell != ' ' for cell in board)


def evaluate(board):
```

```
    win_patterns = [
        (0, 1, 2), (3, 4, 5), (6, 7, 8),  # Rows
        (0, 3, 6), (1, 4, 7), (2, 5, 8),  # Columns
        (0, 4, 8), (2, 4, 6)              # Diagonals
    ]
    for pattern in win_patterns:
        if board[pattern[0]] == board[pattern[1]] == board[pattern[2]]
!= ' ':
            return 10 if board[pattern[0]] == 'X' else -10
    return 0


def get_children(board):
    children = []
    for i in range(len(board)):
        if board[i] == ' ':
            new_board = board[:]
            new_board[i] = 'X' if board.count('X') <= board.count('O')
else 'O'
            children.append(new_board)
    return children


# Example board state for Tic-Tac-Toe
# ' ' represents an empty cell
position = ['X', 'O', 'X',
            ' ', 'X', 'O',
            ' ', ' ', 'O']  # X's turn
depth = 3
is_maximizing = True  # X is maximizing

# Run Minimax with Alpha-Beta Pruning
optimal_value = minimax(position, depth, is_maximizing, float('-inf'),
float('inf'))
print("Optimal value (Minimax with Alpha-Beta):", optimal_value)
```